View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Elsevier - Publisher Connector

Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

Theoretical Computer Science 302 (2003) 401–416

Theoretical Computer Science

www.elsevier.com/locate/tcs

Generalizations of suffix arrays to multi-dimensional matrices

Dong Kyue Kim[a,1], Yoo Ah Kim[b], Kunsoo Park[c,*,2]

[a] *School of Electrical and Computer Engineering, Pusan National University, Pusan 609-735, South Korea*

[b] *Department of Computer Science, University of Maryland, College Park, MD 20742, USA*

[c] *School of Computer Science and Engineering, Seoul National University, Seoul 151-742, South Korea*

Received 28 September 2001; received in revised form 1 October 2002; accepted 21 October 2002
Communicated by A. Apostolico

Abstract

We propose multi-dimensional index data structures that generalize suffix arrays to square matrices and cubic matrices. Giancarlo proposed a two-dimensional index data structure, the Lsuffix tree, that generalizes suffix trees to square matrices. However, the construction algorithm for Lsuffix trees maintains complicated data structures and uses a large amount of space. We present simple construction algorithms for multi-dimensional suffix arrays by applying a new partitioning technique to lexicographic sorting. Our contributions are the first efficient algorithms for constructing two-dimensional and three-dimensional suffix arrays directly.
© 2002 Published by Elsevier Science B.V.

Keywords: Index data structure; Suffix array; Multi-dimensional matrices; Pattern retrieval

1. Introduction

The classical string matching for a pattern p and a text t finds all occurrences of p in t. In many applications [17] ranging from string matching to computational molecular biology, the same text is queried many times with different patterns. Efficient solutions for this problem are based on constructing an index data structure of t that contains an occurrence of p as an *index* in t. Various kinds of index data structures for one-

* Corresponding author.
E-mail address: kpark@theory.snu.ac.kr (K. Park).

[1] This work was supported by Pusan National University Research Grant and partially supported by grant No. R01-2002-000-00589-0 from the Basic Research Program of the Korea Science & Engineering Foundation.

[2] This work was supported by S.N.U. Research Fund 99-11-1-063 and the Brain Korea 21 Project.

0304-3975/03/$ - see front matter © 2002 Published by Elsevier Science B.V.
doi:10.1016/S0304-3975(02)00861-7

dimensional strings have been developed such as suffix trees [5,25,28,30], suffix arrays [24], suffix automata [29], and so on [20].

The *suffix tree* is a compacted tree that represents all suffixes of a text string [25]. It was designed as a space-efficient alternative to Weiner's position tree [30]. A suffix tree for a text t of length n over an alphabet Σ can be built in $O(n \log |\Sigma|)$ time [5,25,28]. We can search a pattern p of length m in $O(m \log |\Sigma|)$ time using the suffix tree. Note that the construction time and the query time depend on the alphabet size. Recently, Farach-Colton et al. [8] gave an $O(n)$-time constructing algorithm for integer alphabets. Although it was mainly designed for pattern matching purposes, the suffix tree is useful for many other applications of string processing [4,7,17,23].

The *suffix array* due to Manber and Myers [24] is basically a sorted list of all the suffixes of a text string and can be constructed in $O(n \log n)$ time [16,24]. When the sorted list is coupled with information about *longest common prefixes* (*lcps*), string searches can be answered in $O(m + \log n)$ time using a simple augmentation to a classic binary search. In practice, suffix arrays use less space than suffix trees, but the construction takes more time [24].

Recently, many algorithms for two- and higher-dimensional pattern matching have been developed. For two-dimensional pattern matching that finds all occurrences of an $m \times m$ pattern P in an $n \times n$ text T, Amir et al. [2] and Galil and Park [9] gave linear-time solutions. For index data structures in two dimensions, Gonnet [15] first introduced a notion of suffix trees for a matrix, called the *PAT-tree*. Giancarlo [10] proposed the *Lsuffix tree* that is a generalization of the suffix tree to square matrices, and gave an $O(n^2 \log n)$-time construction algorithm using $O(n^2)$ space for an $n \times n$ matrix. Giancarlo and Grossi [11] proposed CRCW PRAM algorithms for the construction of Lsuffix trees. Giancarlo and Grossi [12,13] also introduced the general framework of two-dimensional suffix tree families and gave an expected linear-time construction algorithm. In higher dimensions, Giancarlo and Grossi [14] devised generalized index data structures and proposed CRCW PRAM algorithms for the construction.

For two-dimensional suffix arrays, Giancarlo [10] first constructed Lsuffix trees and then obtained two-dimensional suffix arrays from Lsuffix trees. However, the construction of Lsuffix trees maintains complicated data structures and uses a large amount of space, and thus it is not a practical way of constructing two-dimensional suffix arrays. Also, there is no obvious way to extend Manber and Myers's suffix array construction algorithm to two dimensions.

In this paper we propose *Isuffix arrays* and *Zsuffix arrays* that generalize suffix arrays to $n \times n$ square matrices and $n \times n \times n$ cubic matrices, respectively. We first define linear representations of square matrices and cubic matrices. In order to sort the linearly represented suffixes, we develop a new partitioning technique based on Hopcroft's function partitioning [1,18]. By applying the technique, we present a simple and practical algorithm for constructing Isuffix arrays. Our algorithm is independent of the alphabet size and can be easily extended to higher dimensions. Our contributions are the following:

(1) an $O(n^2 \log n)$ time construction algorithm for Isuffix arrays, and
(2) an $O(n^3 \log n)$ time construction algorithm for Zsuffix arrays.

These are the first efficient algorithms that construct multi-dimensional suffix arrays directly.

The paper is organized as follows. In Section 2, we describe a linear representation of square matrices and define two-dimensional suffix arrays. In Section 3, we present a two-phase partitioning technique based on Hopcroft's function partitioning. We present our algorithm for constructing two-dimensional suffix arrays in Section 4. In Section 5 we briefly describe the three-dimensional case and we conclude with some remarks in Section 6.

2. Preliminaries

In this section we first describe a linearization method of square matrices, and then define two-dimensional suffix arrays.

2.1. Linear representation of square matrices

Given an $n \times n$ matrix A, we denote by $A[i:k, j:l]$ the submatrix of A with corners (i,j), (k,j), (i,l), and (k,l). When $i=k$ or $j=l$, we omit one of the repeated indices. An entry of matrix A has a symbol from an alphabet Σ, on which a total order $\prec$ is defined. Consider a string x over alphabet Σ. The ith suffix (resp. prefix) of x is defined as the largest substring of x that starts (resp. ends) at position i. We generalize this definition of suffixes to higher dimensions: For $1 \leqslant i, j \leqslant n$, the *suffix* SA_{ij} *of matrix* A is the largest square submatrix of A that starts at position (i,j) in A. That is, $SA_{ij} = A[i:i+k, j:j+k]$ where $k = n - \max(i,j)$.

Giancarlo [10] proposed two constraints of a two-dimensional index data structure (i.e., completeness and common prefix constraints) so that it can be used for pattern matching purposes when patterns are square matrices. The completeness constraint is that every square submatrix of A must be associated with a *prefix* of a suffix of A and the common prefix constraint is that the same square submatrices of A must be a common *prefix* of some suffixes of A, whatever the definition of *prefix* is. To satisfy these constraints, we adopt a linear representation of a square matrix. Let $I\Sigma = \bigcup_{i=1}^{\infty} \Sigma^i$, where the letter I represents linear shapes. We refer to the strings of $I\Sigma$ as *Icharacters* and we consider each of them as an atomic item. We refer to $I\Sigma$ as the *alphabet of Icharacters*. Two Icharacters are *equal* if and only if they are equal as strings over Σ. Moreover, given two Icharacters Iw and Iu of equal length, $Iw \prec Iu$ if and only if Iw as a string is lexicographically smaller than Iu as a string.

We describe a linearization method for a square matrix $A[1:n, 1:n]$. We linearize the matrix along its main diagonal [3,10]. When we cut a matrix along the main diagonal, it is divided into an upper right half and a lower left half. Let $a(i) = A[i+1, 1:i]$ and $b(i) = A[1:i+1, i+1]$ for $1 \leqslant i < n$, i.e., $a(i)$ is a row of the lower left half and $b(i)$ is a column of the upper right half. Then $a(i)$'s and $b(i)$'s can be seen as Icharacters.

The linearized string IA of matrix $A[1:n, 1:n]$, which is called the *Istring of matrix* A, is the concatenation of Icharacters $IA[1], \ldots, IA[2n-1]$ that are defined as follows: (See Fig. 1.)

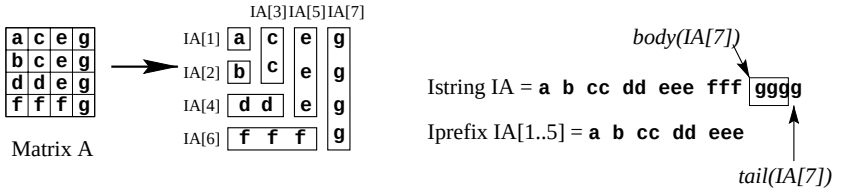


Fig. 1. Istring of a square matrix.

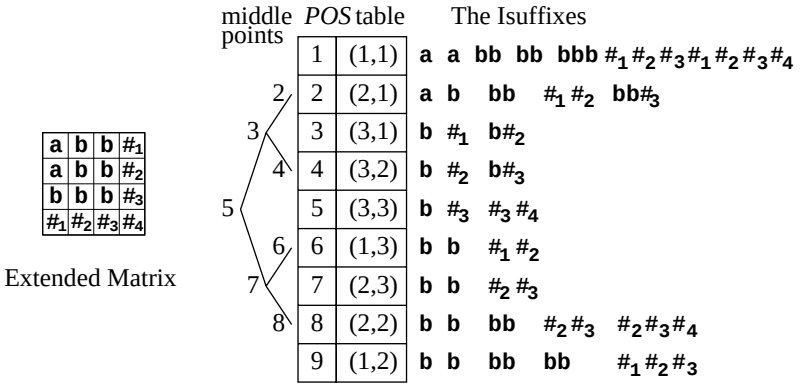


Fig. 2. An Isuffix array of a square matrix.

(i) $IA[1] = A[1,1]$;
(ii) $IA[2i] = a(i)$, $1 \leqslant i < n$;
(iii) $IA[2i+1] = b(i)$, $1 \leqslant i < n$.

Since IA is composed of $2n-1$ Icharacters, the *Ilength* of Istring IA is $2n-1$. The kth *Iprefix* of an Istring IA, denoted by $IA[1..k]$, is the concatenation of Icharacters $IA[1],\ldots,IA[k]$. For each Icharacter $IA[l]$, $1 < l \leqslant 2n-1$, $tail(IA[l])$ is the last character of $IA[l]$ and $body(IA[l])$ is the rest of $IA[l]$. See Fig. 1. Given two Istrings IA and IB, $IA \prec IB$ if IA is smaller than IB in the lexicographic order $\prec$ of Icharacters in IA and IB. The notion of Istrings, where the letter 'I' represents linear shapes, is a simple variant of Lstrings [10], but it plays a crucial role in constructing two-dimensional suffix arrays as well as two-dimensional suffix trees [22].

2.2. Isuffix arrays

Given a text matrix $T[1:n, 1:n]$, we will define Isuffixes of T. Let $\#_i$ be a special symbol not in the alphabet Σ such that $\#_i \prec \#_j \prec a$ for integers $i < j$ and each symbol $a \in \Sigma$. We first define the *extended* matrix $A[1:n+1, 1:n+1]$ of T as follows: (See Fig. 2.)

(i) $A[i,j] = T[i,j]$ for every $1 \leqslant i,j \leqslant n$;
(ii) $A[k,n+1] = A[n+1,k] = \#_k$, for every $1 \leqslant k \leqslant n+1$.

Consider a suffix SA_{ij}, $1 \leqslant i,\ j \leqslant n$, of extended matrix A. The Istring of SA_{ij} is called an *Isuffix* of T and denoted by α_{ij}. Since special symbols were added, there cannot exist a pair of Isuffixes α_{ij} and α_{uv} such that $\alpha_{ij} = \alpha_{uv}$. The number of all Isuffixes of T is n^2.

Now we define two-dimensional suffix arrays, *Isuffix arrays*. The Isuffix array is a suffix array of all the Isuffixes of a given matrix, and consists of three tables *POS*, *Llcp*, and *Rlcp*. The three tables are basically the same as those of Manber and Myers. The basis of the Isuffix array is a lexicographically sorted table *POS*. We define a table $POS[1 : n^2]$ of matrix T as follows: An element $POS[k]$ has the start position (i, j) if and only if α_{ij} is the kth smallest Isuffix in lexicographic order $\prec$. We will construct table *POS* by sorting all the Isuffixes α_{ij} for $1 \leqslant i, j \leqslant n$.

Given two Isuffixes β and γ, let $lcp(\beta, \gamma)$ be the length of the longest common prefix of β and γ when β and γ are regarded as one-dimensional strings. Consider all the possible triples (L, M, R) that can arise in a binary search on the interval $[1 : n^2]$, where L, M, and R denote the left point, middle point, and right point of the interval that remains to be searched. There are exactly $n^2 - 2$ such triples, each with a unique midpoint $M \in [2 : n^2 - 1]$ and we have $1 \leqslant L < M < R \leqslant n^2$ for each triple. Let (L_M, M, R_M) be the unique triple containing midpoint M. *Llcp* and *Rlcp* are tables of size $n^2 - 2$ such that $Llcp[M] = lcp(\alpha_{POS[L_M]}, \alpha_{POS[M]})$ and $Rlcp[M] = lcp(\alpha_{POS[R_M]}, \alpha_{POS[M]})$.

Example 1. In Fig. 2, we give an example of Isuffix arrays. Given a 3×3 text matrix T, we show a table *POS* that is a lexicographic sorted array of all Isuffixes of T. We also show all middle points that can arise in a binary search. If a middle point M is 7, then the unique triple is $(5, 7, 9)$. In this case, we have $Llcp[7] = lcp(\alpha_{3,3}, \alpha_{2,3}) = 1$ and $Rlcp[7] = lcp(\alpha_{2,3}, \alpha_{1,2}) = 2$.

3. Efficient partitioning technique

We will sort the Isuffixes of a matrix using partitioning techniques. Hopcroft [1,18] first proposed the function partitioning technique that takes $O(N \log N)$ time, where N is the input size. Paige et al. [27] gave a linear time solution for the single function partitioning problem. Crochemore [6] used the partitioning technique to find all squares of a string. Iliopoulos et al. [19] also used this technique to find all seeds of a string. Paige and Tarjan [26] gave an algorithm for lexicographic sorting of *independent* strings using a naive partitioning. Gusfield [16] used the function partitioning technique to lexicographic sorting of suffixes of a string. Our partitioning is a two-phase generalization of Hopcroft's function partitioning.

3.1. Equivalence classes

We first define equivalence relations E_l for $1 \leqslant l \leqslant 2n + 1$. E_l is defined on the set of start positions (i, j) of all Isuffixes α_{ij} of matrix T:

$$(i, j) E_l (u, v) \quad \text{if and only if} \quad \alpha_{ij}[1..l] = \alpha_{uv}[1..l].$$

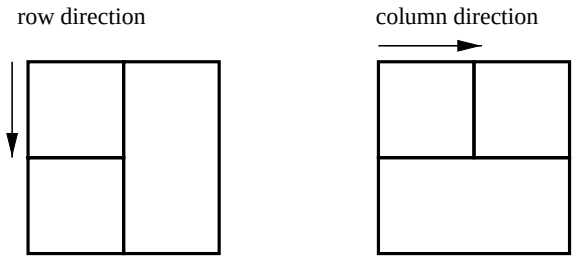


Fig. 3. Doubling the size of submatrices.

That is, (i,j) and (u,v) are in an equivalence class of E_l if and only if the Ilength of the longest common Iprefix between two Isuffixes α_{ij} and α_{uv} is at least l.

To construct one-dimensional suffix arrays, Manber and Myers computed equivalence classes of E_l by doubling the value of l [24]. If we extend their method to two dimensions, we can double the size of submatrices first in the row (or column) direction and then in the other direction. See Fig. 3. However, both of the two ways violate the completeness constraint. For example, none of 3×3 submatrices can be a *prefix* of a suffix. We will extend Gusfield's increment-by-one approach to two dimensions.

We will compute equivalence classes of E_l by increasing the value of l by 1. We first find equivalence classes of E_1 by sorting all symbols of matrix T because each symbol $T[i,j]$ corresponds to the first Icharacter of an Isuffix α_{ij}. The time complexity is $O(n^2 \log n)$ when the alphabet is general. Then, we compute equivalence classes of $E_2, E_3, \ldots$ successively by the partitioning technique until all classes are singleton sets.

At stage l of the partitioning we compute equivalence classes of E_{l+1} from equivalence classes of E_l. To do that, we do not compare $(l+1)$st Icharacters of two Isuffixes, but we use the position information only as follows. We define the *reference* of a position (i,j) in an equivalence class of E_l, denoted by $r(i,j)$: if l is odd, $r(i,j) = (i+1,j)$; if l is even, $r(i,j) = (i,j+1)$. Suppose that $(i,j)E_l(u,v)$. If l is odd, $(l+1)$st Icharacters of Isuffixes α_{ij} and α_{uv} are some subrows in T. In this case, $\alpha_{ij}[l+1] = \alpha_{uv}[l+1]$ if and only if $(i+1,j)E_l(u+1,v)$ (i.e., $r(i,j)E_l r(u,v)$). If l is even, $\alpha_{ij}[l+1]$ and $\alpha_{uv}[l+1]$ are some subcolumns in T. Hence $\alpha_{ij}[l+1] = \alpha_{uv}[l+1]$ if and only if $(i,j+1)E_l(u,v+1)$ (i.e., $r(i,j)E_l r(u,v)$). Therefore, the partitioning is based on

$$(i,j)E_{l+1}(u,v) \quad \text{if and only if} \quad (i,j)E_l(u,v) \quad \text{and} \quad r(i,j)E_l r(u,v).$$

Exploiting this relation directly leads to an $O(n^3)$ time algorithm, since each stage requires $O(n^2)$ time and there are $2n$ stages in the worst case.

We now give some definitions and facts that will be used in the construction algorithm. Consider an equivalence class C of E_l. The *Istring of class C*, denoted by $Istr(C)$, is the common Iprefix of Ilength l of Isuffixes α_{ij}'s for all $(i,j) \in C$. For two equivalence classes C_x and C_y, $C_x \prec_l C_y$ if $Istr(C_x) \prec Istr(C_y)$. Suppose that an equivalence class C of E_l is split into subclasses $C_1, \ldots, C_r$ of E_{l+1} at stage l. The *relative order of class* C_k, $1 \leqslant k \leqslant r$, in $\{C_1, \ldots, C_r\}$, denoted by $order(C_k)$, is the lexicographic order of $Istr(C_k)$ in the set of Istrings $\{Istr(C_1), \ldots, Istr(C_r)\}$. For all

(i,j) in a subclass C_k, $1 \leqslant k \leqslant r$, of C, there must exist one class X_k of E_l such that $r(i,j) \in X_k$. We say that X_k is the *reference class* of C associated with C_k.

Example 2. See Fig. 4. At stage 5, suppose that a class C of E_5 has its Istring $\alpha_{2,1}[1..5]$ and a subclass C_1 of C has its Istring $\alpha_{2,1}[1..6]$. Then the reference class X_1 of C associated with C_1 has its Istring $\alpha_{3,1}[1..5]$.

Fact 1. *Suppose that a class C is split into $C_1,\ldots,C_r$. Let X_k, $1 \leqslant k \leqslant r$, be the reference class of C associated with C_k. If $X_i \prec_I X_j$* then *$C_i \prec_I C_j$* for *$1 \leqslant i, j \leqslant r$. Thus order$(C_k)$ is the relative order of X_k in $\{X_1,\ldots,X_r\}$.*

For each position (i,j) in matrix T, let q_{ij} be the index of the *POS* table such that $POS[q_{ij}] = (i,j)$. We define $rank(C)$ as the minimum of q_{ij}'s for all positions (i,j) in an equivalence class C. Then $rank(C) \leqslant q_{ij} < rank(C) + |C|$ for every $(i,j) \in C$ because all positions in C take contiguous entries in the *POS* table. Notice that for each singleton class D that has one Isuffix α_{uv}, $POS[rank(D)] = (u,v)$.

Fact 2. *Suppose that a class C is split into $C_1,\ldots,C_r$ such that $C_1 \prec_I \cdots \prec_I C_r$. Then $rank(C_1) = rank(C)$ and $rank(C_k) = rank(C_{k-1}) + |C_{k-1}|$ for $2 \leqslant k \leqslant r$.*

Let *Iheight*$[k]$ be the Ilength of the longest common Iprefix of two adjacent Isuffixes $\alpha_{POS[k-1]}$ and $\alpha_{POS[k]}$ for $2 \leqslant k \leqslant n^2$. Fact 3 implies that we can get *Iheight*$[k]$ during the partitioning as a by-product.

Fact 3. *Suppose that a class C is split into $C_1,\ldots,C_r$ at stage l. Then Iheight$[rank(C_k)] = l$ for each C_k, $2 \leqslant k \leqslant r$.*

When we say that we partition a class C *with respect to* a reference class X of C, we partition C into two subclasses C_1 and C_2 such that $C_1 = \{(i,j) \in C \mid r(i,j) \in X\}$ and $C_2 = \{(i,j) \in C \mid r(i,j) \notin X\}$. At stage l of the partitioning we will partition each class C of E_l with respect to reference classes of C.

Example 3. Fig. 4 shows positions in some equivalence classes. Consider an equivalence class $C = \{(2,1),(2,5),(3,9),(9,3),(9,9)\}$ of E_5. Because there exist two distinct reference classes X and X' of C such that $\{(3,1),(3,5)\} \subset X$ and $\{(4,9),(10,3),(10,9)\} \subset X'$, we partition C with respect to X and X'. Thus, C can be split into two subclasses $C_1 = \{(2,1),(2,5)\}$ and $C_2 = \{(3,9),(9,3),(9,9)\}$ of E_6 at stage 5. Since $X \prec_I X'$, we can determine $order(C_1)$ and $order(C_2)$ (i.e., $C_1 \prec_I C_2$) by Fact 1. By Fact 2, $rank(C_1) = rank(C)$ and $rank(C_2) = rank(C_1) + |C_1|$. By Fact 3, $Iheight[rank(C_2)] = 5$.

Consider also class $C_2 = \{(3,9),(9,3),(9,9)\}$ of E_6. Since there are two reference classes Y and Y' of C_2 such that $\{(3,10),(9,4)\} \subset Y$ and $(9,10) \in Y'$, we can partition class C_2 into two subclasses $D_1 = \{(3,9),(9,3)\}$ and $D_2 = \{(9,9)\}$ of E_7. We also have $order(D_1) \prec order(D_2)$, $rank(D_1) = rank(C_2)$, $rank(D_2) = rank(D_1) + |D_1|$, and $Iheight[rank(D_2)] = 6$.

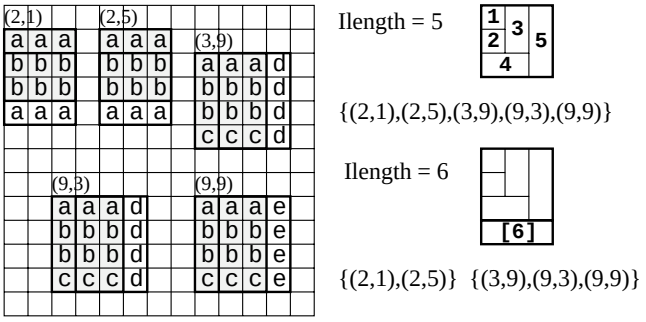


Fig. 4. Positions in equivalence classes.

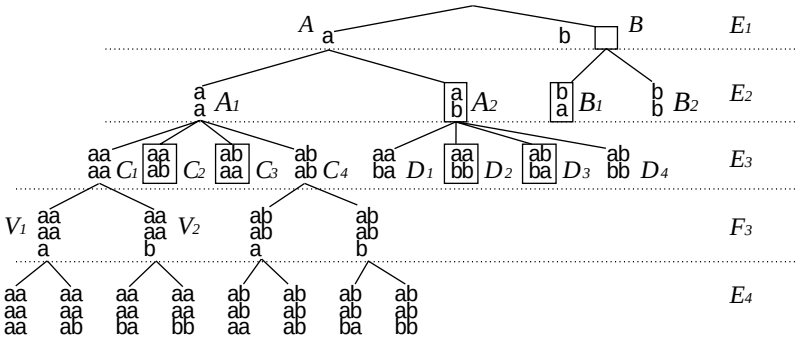


Fig. 5. Istrings of equivalence classes.

3.2. Partitioning technique

We first describe main difficulties when we apply Hopcroft's partitioning technique to sorting the Isuffixes. Suppose that a class C of E_l is partitioned to subclasses $C_1,\ldots,C_r$ of E_{l+1}. We call C the *predecessor* of each C_k, $1 \leqslant k \leqslant r$, denoted by $pred(C_k)$. Among the subclasses $C_1,\ldots,C_r$, a largest one is called a *big* class of E_{l+1} (ties are broken arbitrarily); all the other classes are called *small* classes of E_{l+1}. We will denote the relative order of the big subclass of C by *big-ord*(C). The main idea of Hopcroft's partitioning is to partition with respect to small classes (i.e., with respect to small reference classes in our case). However, we cannot apply this idea to sorting the Isuffixes directly. In order to partition a class C by using the idea, the predecessors of all reference classes of C should be the same. (Otherwise, there may exist two or more reference classes that are not the small classes.) Example 4 shows this case.

Example 4. Fig. 5 shows Istrings of some classes of E_1 through E_4 where $\Sigma = \{a, b\}$. Let A and B be the classes of E_1 such that $Istr(A) = a$ and $Istr(B) = b$. Let A_1 and A_2 be the subclasses of A such that $Istr(A_1) = aa$ and $Istr(A_2) = ab$, and B_1 and B_2 be

the subclasses of B such that $Istr(B_1)=ba$ and $Istr(B_2)=bb$. Suppose that A is big. At stage 1, A_2 is computed by partitioning A with respect to B, and B_2 is computed by partitioning B with respect to B. Then all classes of E_2 are determined.

However, all classes of E_3 cannot be computed by partitioning with respect to the small classes of E_2. Suppose that A_2 and B_1 are the small classes of E_2 and A_1 is split into classes C_1, C_2, C_3 and C_4. Note that A_1, A_2, B_1, and B_2 are the reference classes of A_1, and the predecessors of the reference classes are not the same, i.e., $pred(A_1)=pred(A_2)=A$ and $pred(B_1)=pred(B_2)=B$. When we partition A_1 with respect to small classes A_2 and B_1, we get classes C_2 and C_3 only. To get C_1 and C_4, we should have partitioned A_1 with respect to A_1 or B_2. Hence, at stage $l>1$ partitioning a class of E_l with respect to the small classes of E_l is not sufficient.

To remedy this problem, we define *intermediate* equivalence relations F_l, $1\leqslant l\leqslant 2n$. F_l is also defined on the set of start positions (i,j) of all Isuffixes α_{ij} of matrix T:

$$(i,j)F_l(u,v) \quad \text{if and only if } \alpha_{ij}[1..l]=\alpha_{uv}[1..l] \quad \text{and} \quad body(\alpha_{ij}[l+1])=body(\alpha_{uv}[l+1]).$$

The $(l+1)$st Icharacter of an Isuffix α_{ij} is composed of $body(\alpha_{ij}[l+1])$ and $tail(\alpha_{ij}[l+1])$, which correspond to $\alpha_{r(i,j)}[l-1]$ and $tail(\alpha_{r(i,j)}[l])$, respectively. (See Fig. 1.) Hence, we will use the following relations:

$$(i,j)F_l(u,v) \text{ if and only if } (i,j)E_l(u,v) \text{ and } r(i,j)E_{l-1}r(u,v),$$
$$(i,j)E_{l+1}(u,v) \text{ if and only if } (i,j)F_l(u,v) \text{ and } r(i,j)E_lr(u,v).$$

In the sorting algorithm, we will divide each stage into two phases to perform the partitioning correctly. In the first phase we partition classes of E_l with respect to the small classes of E_{l-1} to get classes of intermediate relation F_l. In the second phase we partition classes of intermediate relation F_l with respect to the small classes of E_l, and we get all classes of E_{l+1}. Suppose that a class D of E_l is split into classes $V_1,\ldots,V_r$ of intermediate relation F_l at the first phase of stage l. For all (i,j) in each intermediate class V_k, $1\leqslant k\leqslant r$, there exist one class Y_k of E_{l-1} such that $r(i,j)\in Y_k$. We call Y_k the 1*st-phase* reference class of D associated with V_k. At the 2nd phase of stage l, suppose that an intermediate class V_k of F_l is again split into classes $C_1,\ldots,C_{r'}$ of E_{l+1}. For all (i,j) in each class $C_{k'}$, $1\leqslant k'\leqslant r'$, there exist one class $X_{k'}$ of E_l such that $r(i,j)\in X_{k'}$. We call $X_{k'}$ the 2*nd-phase* reference class of V_k associated with $C_{k'}$. Note that the reference class of D associated with $C_{k'}$, $1\leqslant k'\leqslant r'$, is the 2nd-phase reference class of V_k associated with $C_{k'}$.

Lemma 1 shows that we correctly partition the classes of E_l with respect to 1st-phase and 2nd-phase reference classes at stage l.

Lemma 1. (1) *Let $Y_1,\ldots,Y_r$ be the 1st-phase reference classes of a class D at stage l. Then $pred(Y_1)=\cdots=pred(Y_r)$.*

(2) *Let $X_1,\ldots,X_s$ be the 2nd-phase reference classes of an intermediate class V at stage l. Then, $pred(X_1)=\cdots=pred(X_s)$.*

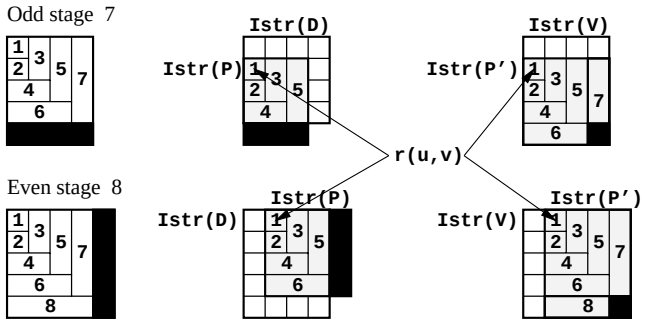


Fig. 6. The Istrings of predecessors of 1st-phase and 2nd-phase reference classes.

Proof. (1) Let P_i be the predecessor of Y_i for each $1 \leqslant i \leqslant r$. Note that P_i, $1 \leqslant i \leqslant r$, is a class of E_{l-2}. In order to prove $P_1 = \cdots = P_r$, we will show that $Istr(P_1) = \cdots = Istr(P_r)$ because an equivalence class has its unique Istring.

Let (x_i, y_i) be a position in class Y_i of E_{l-1} for each $1 \leqslant i \leqslant r$. Then the Istring $Istr(Y_i)$ is $\alpha_{x_i y_i}[1..l-1]$. Since (x_i, y_i) also exists in each predecessor P_i of Y_i, $Istr(P_i) = \alpha_{x_i y_i}[1..l-2]$. By definition of 1st-phase reference class Y_i, there must be a position (u_i, v_i) in D such that $r(u_i, v_i) = (x_i, y_i)$ for all $1 \leqslant i \leqslant r$. See Fig. 6(a). For every position (u, v) in D, $\alpha_{r(u,v)}[1..l-2]$ must be the same since $Istr(D) = \alpha_{uv}[1..l]$. Hence, we have $\alpha_{r(u_1,v_1)}[1..l-2] = \cdots = \alpha_{r(u_r,v_r)}[1..l-2]$, which means that $\alpha_{x_1 y_1}[1..l-2] = \cdots = \alpha_{x_r y_r}[1..l-2]$. Therefore, we get $Istr(P_1) = \cdots = Istr(P_r)$.

(2) Let P'_j be the predecessor of X_j for each $1 \leqslant j \leqslant s$. Similarly, we can show that $Istr(P'_1) = \cdots = Istr(P'_s)$. Let (x_j, y_j) be a position in class X_j of E_l for each $1 \leqslant j \leqslant r$. Then $Istr(X_j) = \alpha_{x_j y_j}[1..l]$. Because P'_j is a class of E_{l-1} and (x_j, y_j) also exists in P'_j, $Istr(P'_j) = \alpha_{x_j y_j}[1..l-1]$. By definition of 2nd-phase reference class X_j, there must be a position (u_j, v_j) in V such that $r(u_j, v_j) = (x_j, y_j)$ for all $1 \leqslant j \leqslant r$. See Fig. 6(b). For every position (u, v) in V, $\alpha_{r(u,v)}[1..l-1]$ must be the same. Hence, we have $\alpha_{r(u_1,v_1)}[1..l-1] = \cdots = \alpha_{r(u_s,v_s)}[1..l-1]$ and $\alpha_{x_1 y_1}[1..l-1] = \cdots = \alpha_{x_s y_s}[1..l-1]$. Therefore, we get $Istr(P'_1) = \cdots = Istr(P'_s)$. □

Example 5. In Fig. 5, we show some classes of F_3 and E_4. The 1st-phase reference classes of C_1 are A_1 and A_2, and $pred(A_1) = pred(A_2) = A$. The 2nd-phase reference classes of V_1 are C_1 and C_2, and $pred(C_1) = pred(C_2) = C$. Similarly, the 2nd-phase reference classes of V_2 are D_1 and D_2, and $pred(D_1) = pred(D_2) = D$.

4. The algorithm for Isuffix arrays

We will describe an algorithm for constructing Isuffix arrays of an $n \times n$ matrix T. Fig. 7 shows MAKE-POS that sorts all Isuffixes of matrix T. MAKE-POS maintains the following invariant:

```
Procedure MAKE-POS
1:    SMALL_0 ← φ;
2:    sort all symbols and make classes of E_1;
3:    determine order, rank, and Iheight for each class of E_1;
4:    set the relative order of the big class C_b of E_1 as big-ord(pred(C_b));
5:    put all classes of E_1 except C_b into SMALL_1;
6:    l ← 1;
7:    while there is a non-singleton class of E_l do
8:        for each class Y of E_{l-1} in SMALL_{l-1} do
9:            partition with respect to Y, every class D of E_l whose 1st-phase reference class is Y;
10:           for each split class D of E_l,
11:               set the relative order of the new intermediate subclass V of D;
12:       od                                                    // 1st-phase partitioning
13:       for each class X of E_l in SMALL_l do
14:           partition with respect to X, every intermediate class V of F_l
                  whose 2nd-phase reference class is X;
15:           for each split intermediate class V of F_l,
16:               set the relative order of the new subclass C of V;
17:       od                                                    // 2nd-phase partitioning
18:       for each split class D in E_l do
19:           for each new subclass C of D do
20:               compute order(C) in D;
21:               determine rank(C) and set Iheight[rank(C)] = l;
22:               if C is a small class then put C into SMALL_{l+1};
23:                 else set big-ord(D) as order(C); fi
24:               if C is singleton class then assign the value of table POS; fi
25:           od
26:       od
27:       l ← l + 1;
28:   od
end
```

Fig. 7. Procedure MAKE-POS that computes table *POS*.

At the beginning of stage l, for all classes C of E_{l-1} *and* E_l, *we know order*(C) *and rank*(C). *All small classes of* E_{l-1} *and* E_l *are in* SMALL_{l-1} *and* SMALL_l, *respectively. For all classes S in* SMALL_{l-1} *and* SMALL_l, *We know big-ord*$(pred(S))$.

When $l=1$, the invariant is satisfied by lines 1–5 of MAKE-POS. In each stage l we compute equivalence classes of E_{l+1}. In the first **for** loop (lines 8–12), we perform the first phase of stage l in order to identify every intermediate class V of F_l and determine the relative order of V. Recall that 1st-phase reference classes at stage l are classes of E_{l-1}. At line 9, we partition with respect to each class Y of E_{l-1} in SMALL_{l-1}, every class D of E_l whose 1st-phase reference class is Y. After we partition with respect to all classes in SMALL_{l-1}, all intermediate classes of F_l can be identified as follows. Let $V_1,\ldots,V_r$ be the intermediate subclasses of a split class D at stage l. Let Y_k, $1\leqslant k\leqslant r$, be the 1st-phase reference classes of D associated with V_k. By definition of small classes and Lemma 1, $r-1$ or r 1st-phase reference classes of D

are small classes depending on whether one or none of $Y_1,\ldots,Y_r$ is big. If none is big then we get every intermediate subclass $V_1,\ldots,V_r$ of D directly (by partitioning with respect to $Y_1,\ldots,Y_r$). Since we know $order(Y_k)$ for all small classes Y_k $(1\leqslant k\leqslant r)$ by the invariant, we can determine the relative order of V_k from $order(Y_k)$ at lines 10–11. Hence, we also get the relative order of intermediate subclasses $V_1,\ldots,V_r$ of D. If one (say, Y_b) is big, then we can compute the intermediate subclass V_b by $V_b = D - (V_1 + \cdots + V_{b-1} + V_{b+1} + \cdots + V_r)$. Moreover, we set the relative order of V_b as $big\text{-}ord(pred(Y_s))$ for a small class Y_s $(1\leqslant s\leqslant r)$. Therefore, we get every intermediate class V of F_l and determine the relative order of V in the first **for** loop.

Similarly, in the second **for** loop (lines 13–17) we perform the second phase of stage l. At line 14, we partition with respect to each class X of E_l in SMALL_l, every intermediate class V of F_l whose 2nd-phase reference class is X. After partitioning with respect to all classes in SMALL_l, we can get every class C of E_{l+1} and determine the relative order of C as in the first phase.

We now describe the third **for** loop (lines 18–26). Suppose that a class D of E_l is split at stage l. For each subclass C of D, since we know the relative order of every intermediate subclass V of D and the relative order of every subclass of V, we can determine $order(C)$ in D. At line 21 we compute $rank(C)$ by Fact 2 and set $Iheight[rank(C)] = l$ by Fact 3. Then, we correctly compute $order(C)$ and $rank(C)$ for all classes C of E_{l+1} at the end of stage l. In order to satisfy the invariant, if C is small then put C into SMALL_{l+1} at line 22; otherwise (i.e., C is big), we set $big\text{-}ord(D)$ as $order(C)$ at line 23. At the end of stage l, every small class S of E_{l+1} are in SMALL_{l+1} and we know $big\text{-}ord(pred(S))$. Therefore, the invariant holds in the next stage. Finally, if C is a singleton subclass that has one Isuffix α_{ij}, we compute the value of table POS: $POS[rank(C)] = (i,j)$.

Lemma 2. MAKE-POS *correctly sorts all Isuffixes of a text matrix* $T[1:n,1:n]$ *and can be implemented in* $O(n^2\log n)$ *time.*

Proof. MAKE-POS is always terminated since there cannot be a pair of Isuffixes α_{ij} and α_{uv} such that $\alpha_{ij} = \alpha_{uv}$. Since the invariant holds for all stages as described, MAKE-POS correctly computes $rank(C)$ for all singleton classes C at line 24. Hence, all elements of the table POS can be determined.

We now consider data structures that are used in MAKE-POS. We implement an equivalence class as a doubly linked list so that insertions or deletions of positions can be done in $O(1)$ time. For each split class D at stage l, we define the *subclass-list* $\mathrm{SUB}_l(D)$ as a linked list sorted by the relative order of all small subclasses of D: An element is associated with a small subclass of D in $\mathrm{SUB}_l(D)$ and points to the doubly linked list that represents the small subclass. We implement SMALL_{l+1} as the set of the subclass-lists $\mathrm{SUB}_l(D)$'s for all split classes D at stage l.

We describe how to implement the partitioning. For each subclass-list $\mathrm{SUB}(Y)$ in SMALL, we extract an element in $\mathrm{SUB}(Y)$ and perform the partitioning with respect to a small subclass X of Y pointed by the element. Notice that, since the elements in $\mathrm{SUB}(Y)$ are sorted by the relative order of subclasses of Y, the relative order of new created classes can be determined by itself. During partitioning with

respect to X, we partition every class D whose reference class is X into two classes $C=\{(i,j)\in D \mid r(i,j)\in X\}$ and $D'=\{(i,j)\in D \mid r(i,j)\notin X\}$. To implement this partitioning, we remove all positions of C from D and insert the deleted positions into a new doubly linked list that represents C. This takes $O(|X|)$ time since a deletion and an insertion can be performed in constant time. Then we insert the pointer of the new doubly linked list into the subclass-list SUB(D). After performing 1st-phase and 2nd-phase partitioning of stage l, we can compute a subclass-list $\mathrm{SUB}_l(D)$ for every split-class D of E_l. SMALL_{l+1} is implemented as the set of $\mathrm{SUB}_l(D)$'s.

We consider the time complexity of the partitioning. By definition of small classes, we have $|C|\leqslant|D|/2$ for each small subclass C of a split class D. Hence one position cannot belong to SMALL_l for some l more than $\log n^2$ times. Since there are n^2 positions, the total number of positions of the classes in SMALL_l for all $1\leqslant l<2n$ is $O(n^2\log n)$. Thus, it takes $O(n^2\log n)$ time to perform the partitioning. The time complexity of the third **for** loop is proportional to the number of classes in SMALL_{l+1} for all $l\geqslant 1$, which cannot exceed $O(n^2\log n)$. Therefore, procedure MAKE-POS can be implemented in $O(n^2\log n)$ time. □

We now construct tables *Llcp* and *Rlcp*. Let *height*[k], $2\leqslant k\leqslant n^2$, be the length of the longest common prefix of $\alpha_{POS[k-1]}$ and $\alpha_{POS[k]}$ as one-dimensional strings. Whenever we determine *Iheight*[*rank*(C)] for each new class C of E_{l+1} in procedure MAKE-POS, *height*[*rank*(C)] can be computed in $O(\log n)$ time using Manber and Myers's *interval trees* [24]. Since an internal node of the interval tree is associated with an interval (L,M,R) that can arise in a binary search, we can directly get tables *Llcp* and *Rlcp* from the interval tree after computing *height*[k] for all $2\leqslant k\leqslant n^2$.

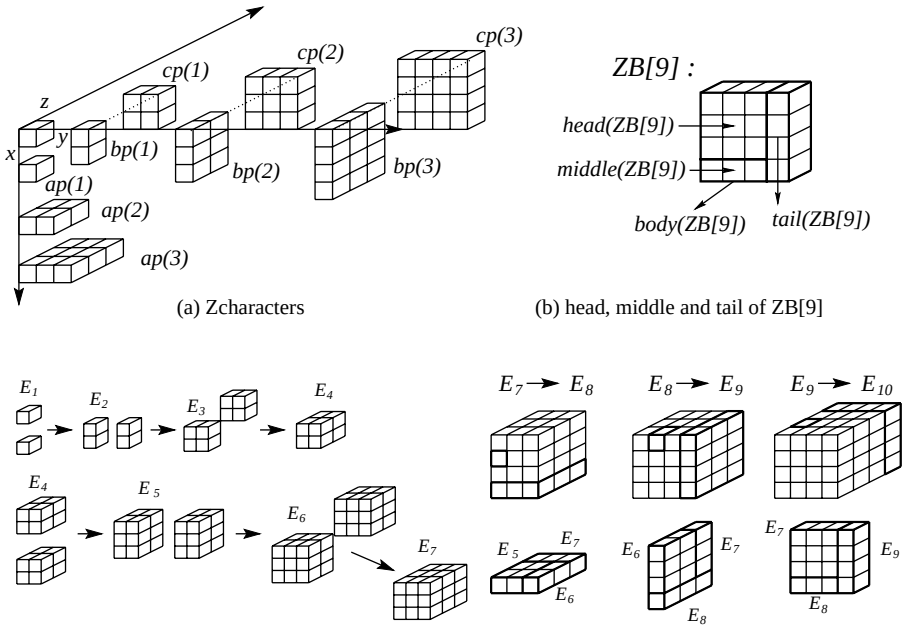
Theorem 1. *The Isuffix array of an $n\times n$ square matrix can be constructed in* $O(n^2\log n)$ *time.*

5. Three-dimensional suffix arrays

We will construct a three-dimensional suffix array, the *Zsuffix array*, which is a generalization of the Isuffix array to three dimensions. In this section, we describe a linearization method of cubic matrices and briefly present how to sort the linearized strings.

Let $Z\Sigma=\bigcup_{i=1}^{\infty}\{\text{all Istrings of Ilength } i\}$. $Z\Sigma$ is called the *alphabet of Zcharacters*, and a linearized string of a cubic matrix $B[1:n,1:n,1:n]$ is called the *Zstring* of B. For $1\leqslant i<n$, a Zstring is composed of three types of planes $ap(i)$, $bp(i)$, and $cp(i)$ that are represented as Istrings and perpendicular to x, y, and z-axis, respectively. The Zstring ZB of matrix B is the concatenation of Zcharacters $ZB[1],\ldots,ZB[3n-2]$ that are defined as follows: (See Fig. 8(a).)

(i) $ZB[1]=B[1,1,1]$;
(ii) $ZB[3i-1]=ap(i)$, where $ap(i)=B[i+1,1:i,1:i]$ of Ilength $2i-1$;
(iii) $ZB[3i]=bp(i)$, here $bp(i)=B[1:i+1,i+1,1:i]$ of Ilength $2i$;
(iv) $ZB[3i+1]=cp(i)$, where $cp(i)=B[1:i+1,1:i+1,i+1]$ of Ilength $2i+1$.



(a) Zcharacters (b) head, middle and tail of ZB[9]

(c) equivalence classes (d) 4x4x4 matrix partitioning

Fig. 8. Zstrings and Equivalence classes of a cubic matrix.

For each Zcharacter $ZB[l]$, $3 < l \leqslant 3n-2$, $tail(ZB[l])$ is the last Icharacter of $ZB[l]$, $body(ZB[l])$ is the rest of $ZB[l]$, $middle(ZB[l])$ is the last Icharacter of $body(ZB[l])$, and $head(ZB[l])$ is the rest of $body(ZB[l])$. See Fig. 8(b). The lth *Zprefix* of an Zstring ZB, denoted by $ZB[1..l]$, is the concatenation of Zcharacters $ZB[1],\ldots,ZB[l]$.

Now we will define three-dimensional suffix arrays, *Zsuffix arrays*. For $1 \leqslant i,j,k \leqslant n$, the *suffix* SB_{ijk} *of* cubic matrix B is the largest cubic submatrix of B that starts at position (i,j,k) in B. For $1 \leqslant i,j,k \leqslant n$, the linearized Zstring of a suffix SB_{ijk} of B, denoted by β_{ijk}, is called a *Zsuffix* of B. Then, the *Zsuffix array* of B can be defined as the suffix array of all Zsuffixes of B.

We define equivalence relations E_l and two intermediate equivalence relations F_l and G_l: E_l, F_l and G_l are defined on the set of start positions (i,j,k) of all Zsuffixes β_{ijk} of a cubic matrix.

$$
\begin{aligned}
&(i,j,k)E_l(u,v,w) \text{ if and only if } \beta_{ijk}[1..l] = \beta_{uvw}[1..l],\\
&(i,j,k)F_l(u,v,w) \text{ if and only if } \beta_{ijk}[1..l] = \beta_{uvw}[1..l] \text{ and}\\
&\qquad\qquad head(\beta_{ijk}[l+1]) = head(\beta_{uvw}[l+1]),\\
&(i,j,k)G_l(u,v,w) \text{ if and only if } \beta_{ijk}[1..l] = \beta_{uvw}[1..l] \text{ and}\\
&\qquad\qquad body(\beta_{ijk}[l+1]) = body(\beta_{uvw}[l+1]).
\end{aligned}
$$

We define the *reference* $r(i,j,k)$ of (i,j,k) in a class C of E_l as follows: if $l \bmod 3 = 1$, $r(i,j,k) = (i+1,j,k)$; if $l \bmod 3 = 2$, $r(i,j,k) = (i,j+1,k)$; if $l \bmod 3 = 0$, $r(i,j,k) =$

$(i,j,k+1)$. The partitioning is based on

$$(i,j,k)E_{l+1}(u,v,w) \text{ if and only if } (i,j,k)E_l(u,v,w) \text{ and } r(i,j,k)E_l r(u,v,w).$$

In Fig. 8(c), the common Zprefixes of Zstrings in equivalence classes of E_1 through E_7 are shown. To apply the partitioning technique, we use the following relations.

$$\begin{aligned}
(i,j,k)F_l(u,v,w) &\text{ if and only if } (i,j,k)E_l(u,v,w) \text{ and } r(i,j,k)E_{l-2}r(u,v,w),\\
(i,j,k)G_l(u,v,w) &\text{ if and only if } (i,j,k)F_l(u,v,w) \text{ and } r(i,j,k)E_{l-1}r(u,v,w),\\
(i,j,k)E_{l+1}(u,v,w) &\text{ if and only if } (i,j,k)G_l(u,v,w) \text{ and } r(i,j,k)E_l r(u,v,w).
\end{aligned}$$

Fig. 8(d) shows how to get equivalence classes of 4^3 cubic matrices using these relations. As in two dimensions, we can construct a Zsuffix array based on these relations.

Theorem 2. *The Zsuffix array of an $n\times n\times n$ matrix can be constructed in $O(n^3 \log n)$ time.*

Corollary 1. *Three-dimensional suffix trees can be constructed in $O(n^3 \log n)$ time.*

6. Concluding remarks

We have defined two- and three-dimensional suffix arrays and presented efficient construction algorithms for them which are based on partitioning techniques. An experiment that compared Isuffix arrays with the original Lsuffix trees [10] was presented in the preliminary version of this paper [21]. According to experimental results, our Isuffix arrays are faster and more space-efficient than Lsuffix trees. These results implies that suffix arrays seem to be more useful index data structures than suffix trees in two and higher dimensions.

Acknowledgements

We are grateful to the referee for his valuable comments to improve the presentation of this paper.

References

[1] A.V. Aho, J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, 1974.
[2] A. Amir, G. Benson, M. Farach, An alphabet independent approach to two-dimensional pattern matching, SIAM J. Comput. 23 (1994) 313–323.
[3] A. Amir, M. Farach, Two-dimensional dictionary matching, Inform. Process. Lett. 21 (1992) 233–239.
[4] A. Apostolico, The myriad virtues of subword trees, in: A. Apostolico, Z. Galil (Eds.) Combinatorial Algorithms on Words, Springer, Berlin, 1985, pp. 85–95.

[5] M.T. Chen, J. Seiferas, Efficient and elegant subword tree construction, in: A. Apostolico, Z. Galil (Eds.), Combinatorial Algorithms on Words, Springer, Berlin, 1985, pp. 97–107.
[6] M. Crochemore, An optimal algorithm for computing the repetitions in a word, Inform. Process. Lett. 12 (1981) 244–250.
[7] M. Crochemore, W. Rytter, Text Algorithms, Oxford University Press, Oxford, 1994.
[8] M. Farach-Colton, P. Ferragina, S. Muthukrishnan, On the sorting-complexity of suffix tree construction, J. Assoc. Comput. Mach. 47 (2000) 987–1011.
[9] Z. Galil, K. Park, Alphabet-independent two-dimensional witness computation, SIAM J. Comput. 25 (1996) 907–935.
[10] R. Giancarlo, A generalization of the suffix tree to square matrices with application, SIAM J. Comput. 24 (1995) 520–562.
[11] R. Giancarlo, R. Grossi, Parallel construction and query of suffix trees for two-dimensional matrices, ACM Symposium on Parallel Algorithms and Architectures, 1993, pp. 86–97.
[12] R. Giancarlo, R. Grossi, On the construction of classes of suffix trees for square matrices: algorithms and applications, Inform. and Comput. 130 (1996) 151–182.
[13] R. Giancarlo, R. Grossi, Suffix tree data structures for matrices, in: A. Apostolico, Z. Galil (Eds.), Pattern Matching Algorithms, Chap. 11, Oxford University Press, Oxford, 1997, pp. 293–340.
[14] R. Giancarlo, R. Grossi, Multi-dimensional pattern matching with dimensional wildcards: data structures and optimal on-line search algorithms, J. Algorithms 24 (1997) 223–265.
[15] G.H. Gonnet, Efficient searching of text and pictures, Tech. Report, University of Waterloo OED-88-02, 1988.
[16] D. Gusfield, An "Increment-by-one" approach to suffix arrays and trees, manuscript, 1990.
[17] D. Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, Cambridge, 1997.
[18] J.E. Hopcroft, An $n \log n$ algorithm for minimizing states in a finite automaton, in: Z. Kohavi, A. Paz (Eds.), Theory of Machines and Computations, Academic Press, New York, 1971, pp. 189–196.
[19] C.S. Iliopoulos, D.W.G. Moore, K. Park, Covering a string, Algorithmica 16 (1996) 288–297.
[20] J. Kärkkäinen, A cross between suffix tree and suffix array, Symposium on Combinatorial Pattern Matching, 1995, pp. 191–204.
[21] D.K. Kim, Y.A. Kim, K. Park, Constructing suffix arrays for multi-dimensional matrices, Symposium on Combinatorial Pattern Matching, 1998, pp. 126–139.
[22] D.K. Kim, K. Park, Linear-time construction of two-dimensional suffix trees, International Colloquium on Automata, Languages and Programming, 1999, pp. 463–472.
[23] G.M. Landau, U. Vishkin, Fast parallel and serial approximate string matching, J. Algorithms 10 (1989) 157–169.
[24] U. Manber, G. Myers, Suffix arrays: a new method for on-line string searches, SIAM J. Comput. 22 (1993) 935–938.
[25] E.M. McCreight, A space-economical suffix tree construction algorithms, J. ACM 23 (1976) 262–272.
[26] R. Paige, R.E. Tarjan, Three partition refinement algorithms, SIAM J. Comput. 16 (1987) 973–989.
[27] R. Paige, R.E. Tarjan, R. Bonic, A linear time solution to the single function coarsest partition problem, Theoret. Comput. Sci. 40 (1985) 67–84.
[28] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (1995) 249–260.
[29] E. Ukkonen, D. Wood, Approximate string matching with suffix automata, Algorithmica 10 (1993) 353–364.
[30] P. Weiner, Linear pattern matching algorithms, Proceedings of the 14th IEEE Symposium on Switching and Automata Theory, 1973, pp. 1–11.